Lojban.class Explanation

The `Lojban` class is the main for the assignments interpreter. This class is responsible for initializing the core components of the interpreter, managing the input loop, and processing each input line.

Core Component Initialization

Upon execution, the `Lojban` class initializes these components:

- **Predicate Database (`predicateDatabase`):** A data structure that stores all defined predicates along with their arguments and associated logic. It is implemented as a `HashMap` where each key is a predicate name and each value is another `HashMap` that maps argument structures (`List<Token>`) to predicate definitions (`Predicate` objects).
- **Lexer (`lexer`):** An instance of the `Lexer` class is created to tokenize user input into a series of tokens that the parser can understand. The lexer abstracts away the details of the textual input, providing a clean, tokenized representation for further processing.
- **Parser (`parser`):** This component uses the list of tokens produced by the lexer into structured `Statement` objects. The parser is aware of the grammar and syntax rules and constructs a parse data structure that represents the user's input semantically.
- **Analyzer (`analyzer`):** It analyzes the parsed statements, applying logical rules and updating the predicate database accordingly. The analyzer is the component where the semantic interpretation of the input occurs.

Error Handling

- The system handles errors by catching `IllegalArgumentExceptions` during processing. If an error occurs, an informative message is displayed, and the program continues, allowing the user to correct or proceed with additional inputs.

Output and Feedback

- After processing each statement, the program outputs the result and a representation of the current state, including the updated predicate database. This immediate feedback helps users understand the effects of their inputs and the interpreter's behavior.

Token.java Explanation

Structure

- **Token Types (`Type` enum):** The `Token` class defines an enumeration `Type` that categorizes the different kinds of tokens that can be recognized in the input. These types correspond to the syntactic elements identified in the assignment prompt:
    - `INITIATOR`: Represents the initiator symbol 'i' that denotes the beginning of a statement.
    - `SHORT_WORD`: Corresponds to short words in Lojban ("lo", "se"), typically a single consonant followed by a vowel.

    - `PREDICATE`: Identifies predicate words (gismu in Lojban), which are central to expressing logical relations and actions.
    - `NUMBER`: Represents numerical literals within the text.
    - `NAME`: Captures named entities, which are indicated by a string enclosed in periods.
    - `LIST`: Signifies a list structure, possibly used in advanced constructs or in representing arguments and their relations.

- **Attributes:**

    - `type (Type)`: This field stores the specific `Type` of the token, indicating its role and classification in the parsed syntax.
    - `value (Object)`: Holds the actual content or value represented by the token. The use of `Object` as the type allows for flexibility in storing various kinds of data, whether they are strings, numbers, or potentially more complex structures for lists or named entities.

Methods

- **Constructor (`Token(Type type, Object value)`):** Initializes a new `Token` instance with a specific type and value, setting the properties that define what the token represents in the context of the input.

- `toString()` **Method:**

    - Provides a string representation of the token, which is particularly useful for debugging and logging. It formats the token's type and value in a readable manner.

- `equals(Object obj)` **Method:**

    - Determines equality between two `Token` instances, considering them equal if they have the same type and value. This method is crucial for comparing tokens, especially when analyzing syntax and structure within the parser.

- `hashCode()` **Method:**

    - Generates a hash code for a `Token` instance, ensuring that tokens with the same type and value receive the same hash code. This is important for the efficient storage and retrieval of tokens, particularly when they are used in collections or need to be uniquely identified.

Statement.java Explanation

The `Statement` class acts as a container for individual statements parsed from the input. Each `Statement` object encapsulates the logical structure defined by the user, specifically the predicate and its arguments.

Structure

- **Predicate (`String predicate`):** This attribute stores the identifier of the predicate, which is the core component of the statement.

- **Arguments (`List<Token> arguments`):** This list holds the tokens that represent the arguments of the predicate. The arguments are essential for the predicate's logic, providing the entities or values

the predicate applies to.

- **Result (`Result result`):** The `result` field captures the outcome after the statement is processed or evaluated. This could be a logical value, a variable, or any other type of result determined by the interpreter's analysis.

Constructor and Methods

- **Constructor (`Statement(String predicate, List<Token> arguments)`):** Initializes a new `Statement` with a specified predicate and a list of arguments. The constructor also initializes the `result` field with a default `null` value, indicating that the statement has not yet been evaluated.
- **`setResult(Result result)` and `getResult()` Methods:**
  - These accessor and mutator methods allow for the manipulation and retrieval of the statement's result. After a statement is analyzed, the result can be set to reflect its outcome, which can then be accessed elsewhere in the program.
- **`toString()` Method:**
  - Provides a string representation of the statement, including its predicate and arguments. This is particularly useful for debugging and output purposes.

Result Class Explanation

Structure

- **Value (`Object value`):** This field holds the actual result of a statement's evaluation. By using an `Object` type, the class is versatile enough to store various types of results, whether they are boolean values, numbers, strings, or even more complex data structures.

Constructor and Methods

- **Constructor (`Result(Object value)`):** The constructor initializes the `Result` object with a specific value. This value is intended to represent the outcome of a statement's evaluation.
- **`getValue()` Method:**
  - Retrieves the encapsulated result value. This method provides access to the underlying value of the `Result`, allowing other components of the interpreter to interpret or further process this value based on the context of the evaluation.
- **`isTrue()` Method:**
  - Determines whether the result represents a "true" value. This method is particularly relevant for logical evaluations, where the truthiness of the result dictates the logical flow or conclusions drawn by the interpreter. The implementation assumes the value is a `Boolean`, which aligns with its primary use case in logical operations.
- **`toString()` Method:**

    - Offers a string representation of the result, which is especially useful for debugging and displaying the output.

Lexer.java Explanation

The `Lexer` class in my implementation serves as the component that breaks down the raw input from the user into a series of tokens. This class is crucial because it abstracts the initial text into a structured format that our `Parser` can easily understand and manipulate.

Tokenization

The `tokenize` method converts a string of input into a list of `Token` objects based on the rules defined for our language syntax. Here's how it operates:

- **Lowercase Conversion:** The input is converted to lowercase to ensure that the tokenization process is case-insensitive, adhering to the requirement that uppercase and lowercase letters are treated equally.
- **Token List Initialization:** A `List<Token>` is initialized to store the sequence of tokens identified in the input.
- **Input Splitting:** The input string is split into parts using whitespace as a delimiter. Each part is examined to determine what type of token it represents.
- **Token Identification:**
    - The method iterates through each split part of the input, applying regex patterns to classify and create the appropriate `Token` objects:
        - `INITIATOR`: Recognizes the initiator symbol 'i', marking the start of a new statement.
        - `SHORT_WORD`: Identifies short words based on the consonant-vowel structure.
        - `NUMBER`: Matches numerical literals, ensuring they don't start with unnecessary leading zeros.
        - `NAME`: Detects names enclosed in periods, capturing variables or specific entities.
        - `PREDICATE`: Identifies predicate words based on their letter patterns (CVCCV or CCVCV).
    - If a part does not match any expected pattern, an `IllegalArgumentException` is thrown, signaling an unrecognized or invalid token.

Parser.java Explanation

The `Parser` class is designed to take a list of tokens generated by the `Lexer` and parse them into structured `Statement` objects. These `Statement` objects encapsulate the logical statements defined by the user, identifying predicates and their corresponding arguments based on the tokenized input.

Constructor and Database Integration

- The `Parser` is initialized with a reference to a `database`, a `HashMap` that stores defined predicates and their logic. This database allows the parser to differentiate between predefined predicates and

potential argument structures during the parsing process.

Parsing Methodology

1. **Initialization:**
    - `statements`: A list to hold all parsed statements.
    - `arguments`: Temporary storage for arguments of the current statement being parsed.
    - `fullArguments`: Captures all tokens related to arguments, aiding in parsing complex structures.
    - `predicate`: Holds the current statement's predicate.
    - `swapNextArguments`: Flags whether the next arguments should be swapped, triggered by the "se" token.
    - `stack`: Supports parsing nested list structures, particularly for handling "lo steko" constructs.
2. **Token Processing:**
    - The parser first ensures the sequence starts with an initiator token, discarding it afterward to prevent false parsing triggers.
    - It iterates over the tokens, applying specific logic based on their type, managed through a series of switch-case statements.
3. **Handling Different Token Types:**
    - **Initiator (`INITIATOR`):** Marks the beginning of a new statement. If there is an existing predicate and arguments, a new `Statement` is formed and added to `statements`. The absence of a predicate triggers an error.
    - **Short Word (`SHORT_WORD`):** When encountering "se," it sets `swapNextArguments` to true. Short words are added to `fullArguments` but not directly to `arguments`.
    - **Number (`NUMBER`):** Validates number usage within context. If not following "lo," it's added to `arguments` through `handleArgumentAddition`, else it triggers an error.
    - **Name (`NAME`):** If following "lo," it's considered a valid name and processed accordingly; otherwise, it might signify a predicate or an error if misused.
    - **Predicate (`PREDICATE`):** Establishes the statement's predicate unless it's an argument-modifying predicate following "lo."
4. **List Handling (`lo steko` and `lo steni`):**
    - For "lo steko," a new list is initiated and managed within `stack` to capture nested list elements.
    - Encountering "lo steni" triggers aggregation of nested lists from `stack` into a final argument structure, reflecting complex, layered data.

Understanding `lo steko` and `lo steni`: - In my parsing logic, `lo steko` initiates a new list context. It signifies the start of a list construction, where subsequent tokens are treated as elements of this list. - `lo steni` signals the termination of a list context. It indicates that all open list constructions (`lo steko`) should be finalized and consolidated into a single structured list.

Example Parsing of `lo steko` and `lo steni`: - Let's consider an example input: `lo steko 1 2 3 lo steko 1 lo steko 2 lo steko 3 lo steni`. - The parser interprets this sequence as follows: - At the first `lo steko`, it begins a new list context, expecting to capture list elements that follow. - The

numbers `1 2 3` are processed and added to this list context. - Upon encountering another `lo steko`, the parser recognizes the start of a new list and whatever follows this `lo steko` is added to this list. So `lo steko 1 lo steko 2 lo steko 3 lo steni` would be `[1],[2],[3]`. - When `lo steni` is encountered, the parser concludes the list construction. It aggregates the collected elements and nested lists, forming a final list representation that encapsulates the entire structure. Therefore, `lo steko 1 2 3 lo steko 1 lo steko 2 lo steko 3 lo steni` yields to `[[1,2,3], [1],[2],[3]]`

5. **Finalization:**
    - Post-iteration, any pending statement with a defined predicate and arguments is finalized and added to `statements`.
    - Errors are thrown for any unresolved or improperly structured syntax, ensuring integrity in the parsing outcome.

Utility Methods

- **`isVariableFollowingLo(List<Token> arguments)`:** Determines whether the next token should be treated as a variable name based on the preceding context (e.g., a name following "lo").
- **`handleArgumentAddition(...)`:** Manages the addition of arguments to the statement, including the logic for the "se" token which requires swapping the next argument with the previous one.

Predicate.java Explanation

The `Predicate` class is the foundation for predicates created from using `cmavo`.

Attributes

- **Name (`String name`):** This attribute stores the identifier for the predicate, which is used to reference it within statements and the broader logical framework of your application.
- **Arguments (`List<Token> arguments`):** This list holds the tokens that represent the arguments of the predicate. These arguments define the entities or values the predicate operates upon or describes.
- **Evaluations (`List<Statement> evaluations`):** This list stores statements that should be evaluated when the predicate is invoked. These statements could represent additional logical assertions or operations that are triggered as part of the predicate's evaluation.

Constructors

- **Default Constructor:** Initializes a predicate with a name but without predefined arguments or evaluations. This constructor sets up a basic predicate structure that can be later enhanced with specific arguments and evaluations.
- **Parameterized Constructor:** Allows for the initialization of a predicate with a full set of attributes, including name, arguments, and evaluations. This is useful for creating more complex predicates that have inherent logic or dependencies encapsulated within them.

Methods

- **`setEvaluations(List<Statement> evaluations)`:** This method allows for the assignment or updating of the evaluations list post-construction.
- **`setArguments(List<Token> arguments)`:** Similar to `setEvaluations`, this method permits the setting or updating of the predicate's arguments.

`toString` Method

- Provides a string representation of the predicate, encapsulating its name, arguments, and evaluations. This method is particularly useful for debugging or logging purposes.

Analyzer.java Explanation

`analyze` Function

Core Functionality:
The `analyze` function serves as the central component of the `Analyzer` class, orchestrating the logical evaluation of a series of parsed statements. It iteratively processes each statement, identifying and executing the corresponding logic based on the predicate each statement contains.

Process Flow:

- The method iterates through the list of provided statements, examining the predicate of each to determine the appropriate handling mechanism.
- Depending on the predicate identified, the function delegates the statement to one of several specialized handler methods (`handleFatci`, `handleSumji`, `handleVujni`, etc.), each tailored to process specific logical constructs.

Predicate Handling:

- For each recognized predicate (`fatci`, `sumji`, `vujni`, `dunli`, `steni`, `steko`, `cmavo`), there is a designated handler method that encapsulates the logic for processing statements with that predicate.
- The switch-case structure facilitates the dispatch of statements to their respective handlers based on the predicate, ensuring each statement is evaluated according to its defined logic.

Fallback Handling:

- If a statement's predicate does not match any of the predefined handlers, it is passed to `handleDatabase`, which processes predicates defined dynamically or stored within the database.

Result Extraction:

- After all statements are processed, the `getLastStatementResult` method retrieves the result of the last statement, which can be particularly useful for interpreting sequences of statements where the outcome of the last operation is of interest.

Helper Method - `getLastStatementResult`:

- This method ensures that the analysis process culminates in the retrieval of the last processed statement's result, providing a concise outcome of the analysis.
- It checks if the statements list is non-empty and returns the last statement. If the list is empty, it returns null, indicating there were no statements to analyze.

Explanation of handleFatci Function

Core Functionality: The `handleFatci` method is specifically tailored to evaluate statements that invoke the 'fatci' predicate. Its primary role is to affirm the existence of the specified entity or concept, ensuring that this assertion is logically recorded within the interpreter's state.

Argument Validation: The method begins by verifying that exactly one argument accompanies the 'fatci' predicate. This validation is crucial because 'fatci' logically requires a single entity for its assertion of existence. If the argument count does not match this requirement, the method throws an `IllegalArgumentException`, signaling a syntax or logical error in the input.

Argument Processing: - The function retrieves the first (and should be only) argument of the statement. - It checks the argument's type to ensure it is either a NAME or a PREDICATE, aligning with the expectation that 'fatci' asserts the existence of named entities or defined predicates. - Assuming the argument passes this validation, the method proceeds to affirm its existence within the interpreter's database.

Database Update: - A new `Predicate` instance is created with the argument's value serving as its identifier. - This predicate is then registered within the interpreter's database, establishing the asserted entity's existence within the system's logical framework. - The registration involves creating an inner mapping (`innerMap`) that associates the argument with the newly created predicate, followed by updating the main database with this association.

Result Update: - After successfully processing the statement, the method updates the statement's result to `true`, reflecting the successful assertion of existence.

Explanation of handleSumji Function

Core Functionality: The `handleSumji` method is specifically designed to manage statements involving the 'sumji' predicate, which is tasked with performing addition operations within the interpreter. It ensures the logical consistency of addition by requiring exactly three arguments, aligning with the predicate's semantics.

Argument Validation: The function enforces strict adherence to argument quantity, demanding three arguments to proceed. Failing this, an `IllegalArgumentException` is raised, signaling improper use of the 'sumji' predicate.

Argument Processing:

- It retrieves the necessary arguments from the statement, ensuring each is of the correct type—either a number or a variable (name).
- The method prepares to execute the addition operation, evaluating the arguments' values and their validity within the given logical context.

Evaluation and Assignment:

- For the execution of the addition operation, the method employs two helper functions: `performOperation` and `assignVariable`.
    - `performOperation` is invoked when the addition needs to be verified against a known sum. It parses the argument values, performs the addition, and confirms the result against the first argument's value, setting the statement's result accordingly.
    - `assignVariable` comes into play when a variable (name) is not present in the environment. Depending on whether subtraction is needed (indicated by the `isVujni` flag), it assigns the result of the addition or subtraction to the variable, updating the environment and the statement's result to reflect this assignment.
    - `assignVariableAdd` is used specifically when addition leads to the assignment of a new variable, updating the environment and the statement's result with the new value.

Error Handling

- Throws an error if at least two arguments are undefined. Meaning we cannot assign nor assert.

Explanation of handleVujni Function

Core Functionality: The `handleVujni` method within my `Analyzer` class is dedicated to managing statements that utilize the 'vujni' predicate, which signifies a subtraction operation. The method ensures that the operation is correctly executed by requiring exactly three arguments, consistent with the predicate's definition.

Argument Validation: The function begins by enforcing the requirement of three arguments. If the number of arguments is not precisely three, it raises an `IllegalArgumentException` to signal a structural issue with the 'vujni' operation.

Argument Processing:

- The method retrieves the three arguments from the statement, preparing to either perform a subtraction operation or carry out variable assignments depending on which argument is unknown and the current state of the environment.

Evaluation and Assignment:

- To execute the subtraction operation, the method relies on the `performOperationSubtract` helper function. This function calculates the difference between the second and third arguments and validates this against the first argument's value, influencing the statement's result.
- Variable assignments are facilitated through the `assignVariable` helper method, which can handle subtraction when the result needs to be stored as a new value in the environment, denoted by a boolean flag indicating the operation type.

Specialized Helper Method:

- The `performOperationSubtract` is an additional helper method specific to the 'vujni' operation. It mirrors the structure of `performOperation` used in the 'sumji' process but is tailored to handle subtraction. This method takes the argument values, performs the subtraction, and sets the statement's result to true or false based on the operation's success.

Error Handling:

- Throws an error if at least two arguments are undefined. Meaning we cannot assign nor assert.

Explanation of `handleDunli` Function

Core Functionality: The `handleDunli` function is designed to assess statements that employ the 'dunli' predicate. It focuses on determining the equality of two arguments, which can be numbers, names, or lists. If an argument is a name not present in the environment, it is assigned the value of the other argument.

Argument Validation: This method commences by asserting the presence of precisely two arguments for the 'dunli' predicate, adhering to its logical requirement of comparing two entities. Should the argument count deviate from two, the method raises an `IllegalArgumentException`, flagging a syntactical or logical discrepancy in the input.

Argument Processing:

- The function retrieves both arguments and ascertains their values using the `getArgumentValue` method.
- Depending on the argument types (number, name, or list), the method handles each scenario distinctly, either by direct assignment (in case one is a name and not in the environment) or by comparison (if both are numbers or strings).

Environment Update:

- If an assignment is required (one argument is a name not in the environment), the function assigns the value of the second argument to the first, reflecting this change in the environment.
- When no assignment is necessary, the method evaluates the equality of the two arguments' values.

Result Update:

- The outcome of the equality check or assignment is encapsulated in a `Result` object, which is then used to update the statement's result.
- If the arguments are equal or an assignment has been successfully made, the result is `true`; otherwise, it is `false`.

Helper Methods Used:

1. `getArgumentValue`: Extracts the value of an argument, handling numbers, names, and lists accordingly.
2. `addAllNestedLists`: A recursive method that flattens nested lists into a single list for comparison or assignment.

Explanation of `handleSteni` Function

Core Functionality: The `handleSteni` method within my `Analyzer` class is specialized to process statements with the 'steni' predicate. This predicate is used to define an empty list, typically for initializing a variable within the logical environment.

Argument Validation: The function checks for the presence of exactly one argument, which is a requirement for the 'steni' predicate. Should the number of arguments be anything other than one, an

`IllegalArgumentException` is thrown, indicating an incorrect usage of the 'steni' predicate.

Argument Processing:

- The sole argument for 'steni' must be a name token, representing the variable to which the empty list will be assigned.
- If the argument is not a name, the function raises an `IllegalArgumentException` to enforce this requirement.

Environment Update:

- The method then proceeds to assign an empty list to the variable indicated by the name token in the environment. This operation effectively initializes the variable with an empty list, setting up its state for potential future additions.

Result Update:

- Following the successful assignment of the empty list, the statement's result is updated with a message detailing the operation, confirming the successful execution of the 'steni' predicate.

Explanation of `handleSteko` Function

Core Functionality: The `handleSteko` method is implemented to process statements involving the 'steko' predicate, which constructs a cons cell in a list. This predicate enables building or extending lists by prepending elements to an existing list or initializing a new list with provided elements.

Argument Validation: The function first verifies that the statement contains either two or three arguments, as required by the 'steko' predicate for proper list construction. An `IllegalArgumentException` is raised if the number of arguments is outside this range, indicating incorrect usage.

Argument Processing:

- The first argument must be a name token, identifying the variable that will reference the new or updated list.
- The second argument represents the head of the list, which may be a direct value or a variable whose value is obtained from the environment.
- If a third argument is provided, it is expected to be a list, which will become the tail of the new list. This list is combined with the head element to form the complete list.

List Construction:

- Utilizing the `addAllNestedLists` helper method, any nested list structures are recursively flattened and added to the new list, ensuring that the list structure is correctly constructed.
- The head element is always added to the new list, and if a tail is provided, it is appended, preserving the order and structure expected in a cons cell.

Environment Update:

- The newly constructed list is then assigned to the variable in the environment using the first argument's name token as the key. This updates the environment to include the new list structure associated with the given variable.

Result Setting:

- Finally, the statement's result is updated to reflect the newly constructed list. This result can be used to confirm the successful execution of the 'steko' predicate and the current state of the list associated with the variable.

Explanation of `handleCmavo` Function

Core Functionality:
The `handleCmavo` method is designed to interpret statements that define new predicates using the 'cmavo' keyword. This function is pivotal for expanding the predicate database within the environment.

Argument Validation:
This function asserts that the statement contains two or three arguments, which aligns with the requirements for defining a predicate. If the number of arguments is incorrect, an `IllegalArgumentException` is raised to signal improper syntax.

Predicate and Argument Association:

- The first argument, expected to be either a name or a predicate, represents the predicate being defined.
- The second argument should be a name or a list of names, representing the associated arguments for the predicate.
- The function verifies these expectations and raises an exception if they are not met.

Nested List Processing:

- If the second argument is a list, the method iteratively processes each nested list to ensure proper association of names to the predicate.
- The `addAllNestedLists` helper method is used to handle nested structures, ensuring that lists within lists are accurately reflected in the arguments.

Evaluations Association:

- If a third argument is present, it is processed to associate evaluation statements with the defined predicate.
- This could involve parsing a list of predicates or a single predicate, depending on the structure of the third argument.

Database Update:

- A new `Predicate` instance is created and added to the database with its arguments and associated evaluations.
- This update to the database effectively registers the new predicate, making it available for future statements.

Result Setting:

- The function concludes by setting the result of the statement to reflect the successful definition of the new predicate.

Explanation of `handleDatabase` Function

Core Functionality:
The `handleDatabase` method is essential for processing statements associated with user-defined or dynamically added predicates. It looks up the predicate in a database of predefined logic and arguments, validating and executing associated logic.

Predicate Verification:
Initially, the method checks if the predicate from the statement exists in the database. If it does not find the predicate, an `IllegalArgumentException` is thrown, indicating the predicate's absence from the known set.

Argument Mapping and Evaluation:

- Once the predicate is confirmed to exist, the method retrieves the associated argument map.
- It checks if the current statement's arguments match any in the argument map. If a match is found, the corresponding `Predicate` object's evaluations are processed.
- If evaluations exist, they are analyzed recursively using the `analyze` method to ensure nested logic is fully evaluated.

Result Assessment:

- After processing evaluations, the method assesses their outcomes. If all evaluations return `true`, the statement's result is set to `true`; otherwise, it is set to `false`.
- If no direct argument match is found, the method attempts to find a valid argument pattern or placeholder match using `findMatchingArgument`.

Placeholder Handling:

- `findMatchingArgument` seeks potential argument matches, considering placeholders for dynamic argument evaluation.
- If placeholders or valid patterns are identified, they inform the statement's result, establishing a logical connection based on available information.