Lojban Predicate Calculus Interpreter: Implementation and Explanation

Noah deMers

October 26 2025

Contents

1 Overview

This report explains the implementation of a predicate calculus interpreter using a subset of Lojban syntax. Lojban is a constructed logical language designed for unambiguous communication. The interpreter implements basic logical operations, arithmetic, list manipulation, and user-defined predicates.

1.1 Assignment Requirements

The assignment is worth 40 points distributed as follows:

- **Scanning & Parsing**: 12 points (6 implementation + 6 explanation)
- **Short Words** (lo, se): 2 points (1 point each)
- **Predicate Words**: 16 points (7 predicates + 9 points explanation)
- **Extra Credit**: Enhanced features beyond requirements

1.2 Architecture Overview

The interpreter follows a four-phase architecture:

1. **Lexical Analysis**: Character classification and tokenization
2. **Syntax Analysis**: Statement parsing and validation
3. **Semantic Analysis**: Type checking and variable binding
4. **Execution**: Statement evaluation and result computation

2 Scanning and Parsing (12 points)

2.1 Lexical Analysis (6 implementation + 6 explanation)

2.1.1 Character Classification

The scanner must distinguish four types of words based on character patterns:

Short Words (cmavo) follow the CV pattern: exactly one consonant followed by one vowel. Examples include `lo`, `se`, and `ko`. The implementation checks that the word length is exactly 2, the first character is a consonant, and the second is a vowel.

Predicate Words (gismu) are exactly 5 letters matching either CVCCV or CCVCV patterns. The CVCCV pattern is consonant-vowel-consonant-consonant-vowel (like `fatci`), while CCVCV is consonant-consonant-vowel-consonant-vowel (like `broda`). The implementation checks both patterns using OR logic to accept either form.

Numbers must be valid integers without leading zeros. The exception is the number `0` itself, which is valid. This prevents ambiguous interpretations like octal literals. Numbers like `007` are rejected, while `0`, `42`, and `1003` are accepted.

Names are variables enclosed in periods, like `.brook.` or `.answer.`. The pattern requires a minimum length of 3 (two periods plus at least one letter), periods at both start and end, and only alphabetic characters between the periods.

2.1.2 Token Validation

Every token in the input must match one of the four allowed patterns. The validation process checks each pattern in sequence and raises a descriptive error if no pattern matches. This provides clear feedback about invalid input, helping users identify syntax errors quickly.

2.1.3 Error Handling Design

The error handling strategy emphasizes clarity and usefulness. When validation fails, the error message includes the invalid token and indicates which patterns were attempted. For example, `"Invalid token: 'xyz'"` immediately tells the user what failed without requiring deep knowledge of the grammar.

2.2 Statement Parsing

2.2.1 Tokenization Process

The input string is split into whitespace characters (spaces, tabs, newlines). All tokens are converted to lowercase for case-insensitive processing as specified in the assignment. The amount and type of whitespace do not matter—only its presence as a token separator.

2.2.2 Statement Boundaries

Statements are delimited by the letter `i`. The parser accumulates tokens into a current statement buffer until it encounters another `i`, at which point it saves the accumulated statement and starts a new buffer. This allows multiple statements on one line or spread across multiple lines.

For example, these are equivalent:

```
i lo .a. fatci   i lo .b. fatci
```

```
i lo .a. fatci
i lo .b. fatci
```

2.2.3 Multi-Statement Programs

The parser returns a list of statement token lists, preserving order for sequential execution. Each statement is evaluated in order, with later statements able to reference variables bound by earlier ones. This enables programs to build up complex logical relationships through a series of simple assertions.

3 Short Words (cmavo) - 2 points

3.1 lo - "the" (1 point)

3.1.1 Purpose and Function

The word `lo` serves as a definite article marker in Lojban. In our implementation, it signals that the following token should be treated as a specific, identifiable entity rather than a general class.

3.1.2 Three Uses

Before Names: `lo .brook.` refers to the specific variable named "brook". Without `lo`, the parser wouldn't know to treat `.brook.` as a variable reference.

Before Predicates: `lo fatci` uses the predicate `fatci` as a noun—the concept of existence itself rather than the act of declaring existence.

Before Lists: `lo steko` begins a list construction, indicating we're building a specific list structure.

3.1.3 Implementation Strategy

The parser recognizes `lo` tokens and sets a flag indicating the next token should receive special treatment. After noting the flag, `lo` is skipped, allowing the semantic token to be processed directly. This maintains clean separation between syntax markers (like `lo`) and semantic content (like variable names).

3.2 se - Argument Swapper (1 point)

3.2.1 Purpose

In Lojban, `se` provides word order flexibility by swapping the first and second arguments of predicates. This allows the same predicate to be expressed in multiple ways depending on what the speaker wants to emphasize.

3.2.2 Swap Semantics

Consider addition: normally we write `result sumji operand1 operand2` meaning result = operand1 + operand2.

With `se`, we can write `se operand1 sumji result operand2`, which swaps the first two arguments so that operand1 and result trade places.

Example: `i lo .answer. sumji 40 2` binds answer to 42. Equivalently, `i se 40 sumji lo .answer. 2` produces the same result by swapping the positions.

3.2.3 Argument Count Handling

For predicates with only one argument, `se` has no effect (there's nothing to swap with), but it's not an error. For predicates with three or more arguments, only the first two are swapped—later arguments remain in their original positions. This behavior matches Lojban's linguistic design where `se` is a binary operator affecting only adjacent argument positions.

4 Predicate Words - 16 points

4.1 fatci - Exists (1 point implementation + 1 point explanation)

4.1.1 Meaning

`fatci` declares that something exists. In Prolog terms, it asserts a fact into the knowledge base.

4.1.2 Behavior

When applied to a variable name like `.brook.`, `fatci` binds that variable to `True` in the variable database. When applied to a number like `42`, it simply returns `True` (numbers exist trivially).

Example: `i lo .brook. fatci` is equivalent to Prolog's `brook.`—it declares the fact that brook exists.

4.1.3 Return Value

`fatci` always returns `True` because existence assertions either declare new facts (which succeeds) or state obvious truths (like numbers existing).

4.2 sumji - Plus (1 point + 1 point explanation)

4.2.1 Meaning

`sumji` performs addition with the pattern: `result sumji operand1 operand2` where result = operand1 + operand2.

4.2.2 Three Operational Modes

Mode 1 - Checking: When all three arguments have values, `sumji` verifies the equation. Example: `i 4 sumji 2 2` checks if 4 = 2 + 2, returning `True`.

Mode 2 - Computing Forward: When the result is unbound but operands have values, `sumji` computes and binds the result. Example: `i lo .answer. sumji 40 2` binds answer to 42.

Mode 3 - Verifying Consistency: If the result was previously bound to a different value, `sumji` checks consistency. If `.answer.` was already 50, then `i lo .answer. sumji 40 2` would return `False`.

4.2.3 Error Conditions

If either operand is an unbound variable, `sumji` raises an error because we can't add unknown values. The result can be unknown (we'll compute it), but the operands must be known.

4.3 vujni - Minus (1 point + 1 point explanation)

4.3.1 Meaning

`vujni` performs subtraction with the pattern: `result vujni operand1 operand2` where result = operand1 - operand2.

4.3.2 Structural Similarity to sumji

`vujni` has identical operational modes to `sumji`: checking, computing forward, and verifying consistency. The only difference is the arithmetic operation (subtraction instead of addition).

Examples:

- `i lo .diff. vujni 10 3` binds diff to 7
- `i 5 vujni 8 3` verifies 5 = 8 - 3, returns `True`
- `i 60 vujni 65 lo .x.` binds x to 5 (since 65 - 5 = 60)

4.3.3 Swap Interaction

Like `sumji`, `vujni` respects the `se` modifier. When `se` is present, the first two arguments swap before computation.

4.4 dunli - Equal (1 point + 1 point explanation)

4.4.1 Meaning

`dunli` checks equality between two values. Pattern: `arg1 dunli arg2`

4.4.2 Key Difference from Arithmetic Predicates

Unlike `sumji` and `vujni`, `dunli` doesn't bind variables—it only returns true or false. It's a pure comparison operator.

4.4.3 Type Flexibility

`dunli` works for any value type: numbers, names, or lists. The equality is determined by the `__eq__` methods defined on each value class.

Examples:

- `i 5 dunli 5` returns `True`
- `i lo .x. sumji 2 3 i lo .x. dunli 5` returns `True`
- `i lo .a. dunli lo .b.` compares variables a and b

4.4.4 Unbound Variables

If either argument is unbound, `dunli` returns `False` (you can't compare an unknown value). This is different from raising an error—we simply say "these values are not equal" when we can't verify equality.

4.5 steni - Empty List (1 point + 1 point explanation)

4.5.1 Meaning

`steni` represents the empty list, analogous to `nil` in Lisp or `[]` in Python.

4.5.2 Two Roles

As Assignment: `i lo .nothing. steni` binds the variable "nothing" to the empty list `()`.

As Literal: `lo steni` represents the empty list as a value in expressions. For example, `i lo steni fatci` declares that the empty list exists, returning `True`.

4.5.3 List Terminator

In list construction with `steko`, `steni` marks the end of the list. The pattern `lo steko item1 lo steko item2 lo steni` builds a two-element list terminated by the empty list.

4.6 steko - Cons Cell / List (1 point + 1 point explanation)

4.6.1 Meaning

`steko` constructs lists using cons-cell structure (head-tail pairs), similar to Lisp's `cons`.

4.6.2 Pattern

`result steko head tail` creates a cons cell where `head` is the first element and `tail` is the rest of the list.

4.6.3 List Building Examples

Single Element: `lo steko 1 lo steni` creates the list `(1)`. Here 1 is the head, and `steni` (empty list) is the tail.

Two Elements: `lo steko 1 lo steko 2 lo steni` creates `(1 2)`. The outer `steko` has head 1 and tail `(2)`, where `(2)` itself is a `steko` with head 2 and tail `()`.

Three Elements: `lo steko 1 lo steko 2 lo steko 3 lo steni` creates `(1 2 3)`. This nests three cons cells.

4.6.4 Recursive Structure

The parser recursively processes nested `steko` expressions, accumulating items until it encounters `steni`. This allows natural expression of arbitrary-length lists while maintaining the cons-cell semantic structure.

4.6.5 Variable Assignment

`i lo .list. steko 1 lo steko 2 lo steni` binds the variable "list" to `(1 2)`. Later code can reference `lo .list.` to access this list.

4.7 cmavo - Predicate Definition (1 point + 1 point explanation)

4.7.1 Meaning

`cmavo` defines new predicates, similar to Prolog clauses. Pattern: `predicate_name cmavo parameters body`

4.7.2 Three Components

Predicate Name: Either a name word like `lo .parent.` or a predicate word like `pinxe`. This becomes the identifier for invoking the predicate.

Parameters: A list of names representing the predicate's arguments. Uses `steko` syntax: `lo steko lo .X. lo steko lo .Y. lo steni` for two parameters X and Y.

Body: A list of predicates to evaluate when this predicate is invoked. Empty body (`lo steni`) means the predicate is a simple fact. Non-empty body means the predicate has rules.

4.7.3 Example: Simple Fact

The Prolog statement `parent(Brook, George).` translates to:

```
i lo .parent. cmavo
  lo steko lo .Brook. lo steko lo .George. lo steni
  lo steni
```

Breaking this down:

- `lo .parent.` is the predicate name
- `lo steko lo .Brook.  lo steko lo .George.  lo steni` is the parameter list (Brook, George)
- `lo steni` is the empty body (no further actions)

4.7.4 Example: Rule

The Prolog rule `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).` would have:

- Parameters: `(X Y)`
- Body: list containing two predicate calls: `X parent Z` and `Z parent Y`

4.7.5 Storage and Retrieval

Defined predicates are stored in a dictionary keyed by predicate name. When a statement invokes a user-defined predicate, the interpreter looks it up in this dictionary and evaluates it according to its parameter bindings and body.

5 Testing and Validation

5.1 Test Coverage

The implementation includes comprehensive tests demonstrating all required features:

Test 1 - fatci: Declares a variable exists, verifying variable binding.

Test 2 - sumji checking: Verifies arithmetic with known values (4 = 2 + 2).

Test 3 - sumji assignment: Computes and binds a variable (answer = 40 + 2).

Test 4 - se modifier: Tests argument swapping with arithmetic.

Test 5 - vujni: Tests subtraction (diff = 10 - 3).

Test 6 - dunli: Tests equality checking after computation.

Test 7 - Multiple statements: Tests sequential execution with variable dependencies.

5.2 Error Handling

The implementation provides clear error messages for common problems:

Invalid tokens: "Invalid token: 'xyz'" when input doesn't match any pattern.

Missing arguments: "sumji requires 3 arguments" when predicate is called incorrectly.

Unbound operands: "Operands must have values" when arithmetic attempted on unknowns.

Unknown predicates: "Unknown predicate: badname" when referencing undefined predicates.

6 Extra Credit Features

6.1 Debug Mode

The interpreter includes a debug flag that traces execution in detail. When enabled, it shows:

- Variable bindings as they occur
- Statement evaluation results
- Predicate definition registration

This helps users understand how their programs execute and diagnose unexpected behavior.

6.2 Extensible Architecture

The predicate system is fully extensible beyond the required predicates:

- Users can define predicates with arbitrary gismu or name patterns
- Supports 1-5 parameters as specified
- Bodies can contain unlimited predicates (no hard-coded limit)
- Recursive predicates are supported structurally

6.3 Type System

The value type hierarchy (LojbanNumber, LojbanName, LojbanList, LojbanPredicate) provides:

- Type safety through dataclasses
- Clean equality semantics
- Polymorphic operations
- Easy extension for new value types

7 Design Rationale

7.1 Architectural Choices

Eager Evaluation: The interpreter evaluates statements immediately rather than building an abstract syntax tree. This simplifies implementation while maintaining correctness for the specified subset.

Dictionary-Based Storage: Variables and predicates use dictionaries for O(1) lookup. This scales well for typical program sizes without unnecessary complexity.

Immutable Semantics: Once bound, variables maintain their values throughout execution. This matches logical programming semantics where facts don't change after assertion.

7.2 Trade-offs

Simplicity vs. Features: The implementation prioritizes correctness and clarity over advanced features like backtracking or constraint propagation. This makes the code easier to understand and maintain.

Error Handling vs. Flexibility: Strict validation catches errors early but rejects some potentially valid constructions. This trade-off favors reliability over permissiveness.

8 Conclusion

This implementation successfully creates a working predicate calculus interpreter using Lojban-inspired syntax. All required features are implemented correctly:

- **Scanning & Parsing**: Four token types validated, statements parsed correctly
- **Short Words**: `lo` and `se` with proper semantics
- **Predicates**: All seven predicates (fatci, sumji, vujni, dunli, steni, steko, cmavo) working correctly
- **Extra Features**: Debug mode, extensible architecture, clean type system

The implementation demonstrates understanding of:

- Lexical analysis and pattern matching
- Parsing with grammar rules
- Variable binding and scope
- Predicate logic semantics
- List data structures
- Error handling and validation