Developer Reference Manual

1. Internal Architecture Overview

The Lexis interpreter follows a modular architecture divided into four primary subsystems:

1. **Lexical Analysis Subsystem**
   Converts input strings into tokens.
2. **Parsing Subsystem**
   Produces an Abstract Syntax Tree (AST) structure from tokens.
3. **Execution Subsystem (Core Interpreter)**
   Performs the semantic execution of commands and maintains global program state.
4. **Frontend Interfaces**
   The REPL and web API forward user commands into the interpreter pipeline.

Each subsystem operates independently, with communication defined through structured interfaces (tokens, AST nodes, and execution outputs).

2. Internal Data Structures and State Management

The interpreter maintains several internal attributes that define the current state of the program. These values persist across commands and determine execution outcomes.

2.1. Mode State

- **mode**

  Indicates whether the interpreter is in Play Mode or Edit Mode. Affects which command set is permitted and how the parser interprets commands.

- **file_mode**
  Determines the structure of the active word bank.
  Values: "letters", "hints", "categories".

2.2. Active File State

- **current_filename**

  Path of the currently selected word bank file.
- **words**

  List of words (Letters Mode).
- **word_data**

  List of lists, each containing a word followed by hints or category values.
- **categories**

  List of category names (Categories Mode only).

2.3. Game Runtime State

- **secret_word**

  The selected answer for Play Mode.
- **max_guesses**

  Maximum allowed guesses for the current session.
- **remaining**

  Remaining guesses.
- **history**

  Stores previous guesses and their feedback outputs.

2.4. Invariants

The following invariants must hold throughout execution:

1. current_filename must reference an existing file after load operations.
2. In Categories Mode, all entry lists must match the length of categories.
3. word_data rows must contain consistent field counts within their mode.
4. remaining must not exceed max_guesses.
5. secret_word must be selected from the currently loaded word bank.

Violations of these invariants may result in inconsistent gameplay or incorrect output.

3. AST Node Reference

Each AST node corresponds to a single command recognized by the parser. Nodes hold parameters extracted from input and are consumed by command handlers in the interpreter. Below is the developer-oriented reference listing all nodes, their fields, and their associated execution function.

3.1. Play Mode Nodes

Start

- **Fields**: none
- **Handled by**: _start_game()
- **Purpose**: Initializes game state using current word bank.

Guess

- **Fields:** word
- **Handled by:** _guess()
- **Purpose:** Compares the guessed word with the secret word and updates state.

MaxGuesses

- **Fields:** count
- **Handled by:** _set_max_guesses()
- **Purpose:** Updates guess limit; affects remaining attempts.

File

- **Fields:** filename
- **Handled by:** _load_file()
- **Purpose:** Loads a word bank and switches file_mode accordingly.

Edit

- **Fields:** optional filename
- **Handled by:** _switch_to_edit_mode()
- **Purpose:** Transitions interpreter state to Edit Mode.

Quit

- **Fields:** none
- **Handled by:** _quit()
- **Purpose:** Ends current session (REPL-level behavior).

Help

- **Fields:** none
- **Handled by:** _help_play()
- **Purpose:** Outputs Play Mode command list.

3.2. Edit Mode Nodes

Create

- **Fields:** filename
- **Handled by:** _create_file()

DeleteFile

- **Fields:** filename
- **Handled by:** _delete_file()

Categories

- **Fields:** list_of_category_names
- **Handled by:** _set_categories()

Add

- **Fields:** word, values
- **Handled by:** _add_word()

Edit

- **Fields:** word, new_values
- **Handled by:** _edit_word()

Delete

- **Fields:** word
- **Handled by:** _delete_word()

ListWords

- **Fields:** none

- **Handled by:** _list_words()

Done

- **Fields:** none
- **Handled by:** _switch_to_play_mode()

Help

- **Fields:** none
- **Handled by:** _help_edit()

4. Interpreter Function Reference

This section explains the internal behavior of major interpreter functions, focusing on side effects, state transitions, and required invariants.

4.1. File Handling Functions

_load_file(filename)

Inputs: filename
Side Effects:

1. Reads file and parses mode (letters/hints/categories).
2. Populates words, word_data, categories.
3. Updates file_mode.
4. Resets game runtime state.

Failure Conditions:

- Missing file
- Malformed line structure
- Inconsistent category counts

_save_file()

Inputs: none
Side Effects:

1. Rewrites entire file according to current file_mode.

2. Normalizes spacing and removes malformed entries.

Developer Notes:

- This function overwrites all existing formatting.
- Category mode requires fixed-length rows.

4.2. Edit Operations

_add_word(node)

Updates word bank based on mode:

- Letters → append word to words.
- Hints → append [word, hint1, hint2, …].
- Categories → append [word, cat1_val, cat2_val, …].

Failure Conditions:

- Duplicate word entries
- Value length mismatch in Categories Mode

_edit_word(node)

Modifies existing word entry.

Effects:

- Locates matching entry
- Updates values
- Respects category field count

_delete_word(node)

Removes word entry from the active bank.

Effects:

- Adjusts internal lists

- Ensures no partial deletion occurs

_set_categories(node)

Initializes category labels for Categories Mode.

Effects:

- This resets all existing category data
- Must be executed before adding any words

4.3. Play Mode Execution

_start_game()

Randomly selects secret_word and resets gameplay state.

Effects:

- history cleared
- remaining set to max_guesses
- secret_word assigned

_guess(node)

Evaluates guessed word depending on mode.

Shared behavior across modes:

- Decrements remaining
- Records guess in history
- Determines end-of-game conditions

Mode-Specific Behavior:

- Letters → checks character accuracy
- Hints → requires strict membership in bank
- Categories → compares category values

5. Word Bank Processing

5.1. Mode Detection Algorithm

The interpreter determines file mode using the following logic:

1. If the first line contains multiple pipe-separated values → Categories Mode.
2. Otherwise, scan lines:
    - If any line contains pipes → Hints Mode.
    - If no pipes exist → Letters Mode.

Mode determination affects all subsequent parsing and saving.

5.2. Internal Storage Rules

- Letters Mode: entries stored as simple strings.
- Hints Mode: stored as lists of [word, hint1, hint2, ...].
- Categories Mode: stored as [word, val1, val2, ...] with strict length checks.

6. Execution Flow

The internal pipeline operates as:

1. Lexer.lex() produces a token list.
2. Parser.parse() consumes tokens and outputs an AST node.
3. Interpreter.execute() performs a type check on the node.
4. A command-specific handler is invoked.
5. Interpreter updates internal state.
6. Output is returned upward to REPL or API.

This structured flow enables reliable debugging and clear responsibility boundaries.

7. Mode Switching Rules

7.1. Switching into Edit Mode

Triggered by Edit Mode AST node.

Effects:

- mode = "edit"

- Parser must be reconstructed for Edit Mode commands.

7.2. Switching into Play Mode

Triggered by the Done node.

Effects:

- mode = "play"
- Existing file remains loaded
- Parser must switch back to Play Mode grammar

7.3. Developer Constraints

- State must remain consistent during mode transitions
- Word bank must be loaded before starting play
- Categories list must match row widths

8. Error Handling Model

Lexis uses a dual-layer error system.

8.1. Developer-Level Exceptions

Used only for internal issues:

- ParserError
- InterpreterError

These indicate an implementation or structural problem, not user error.

8.2. User-Level Errors

Returned as plain strings:

- “Error: word not found”
- “Error: invalid format”
- “Error: file does not exist”

Frontends handle display formatting.

9. Extending the Interpreter

9.1. Adding a New Command

To introduce a new command:

1. Define an AST node class in play.py or edit.py.
2. Add a parser rule and keyword entry.
3. Implement the corresponding interpreter handler.
4. Ensure state changes remain consistent.
5. Update help text in the respective mode.

9.2. Adding a New Game Mode

Requires:

1. A new file format definition
2. A new detection rule in file loading
3. Mode-specific storage logic
4. Mode-specific guessing logic
5. Updated save behavior

Ensure each mode's data structures remain mutually exclusive and consistent.

10. Known Technical Limitations

1. Word bank parsing lacks strict schema validation.
2. interpreter.py handles many responsibilities; module separation can improve maintainability.
3. Category Mode requires careful value-length matching.
4. File rewrites overwrite original formatting completely.
5. REPL and API duplicate some behavior and may benefit from shared utilities.

11. Recommended Refactor Plan

1. Split interpreter.py into separate modules (file handling, game logic, edit logic).
2. Introduce validation utilities for file formats.
3. Implement unified error objects for cleaner frontend handling.
4. Add unit tests for lexer, parser, and interpreter subsystems.

5. Replace lists with dataclasses for structured entries.