Architecture for Integrating a Lisp Interpreter with an SVG 1.1 Renderer

May 17, 2025

Contents

1 Introduction

This document outlines an abstract architecture for integrating a Lisp interpreter with an SVG 1.1 renderer to enable Lisp instructions to drive vector graphics rendering directly to images. The renderer uses SVG-like abstractions (e.g., coordinates, strokes, fills) but produces images (e.g., PNG, JPEG) without generating SVG XML. The Lisp interpreter, structured as a modular system, executes graphics commands that map to renderer operations, providing a high-level interface for vector graphics.

The architecture preserves the interpreter's modularity, isolates renderer-specific logic, and ensures a natural mapping between Lisp's functional paradigm and the renderer's vector graphics model. This document focuses on the high-level design, component interactions, and data flow, avoiding implementation details.

2 Architecture Overview

The architecture integrates the Lisp interpreter and SVG renderer through three layers:

1. **Lisp Interpreter Core**: Parses and evaluates Lisp code, unchanged from its existing modular design.
2. **Graphics Interface**: Defines Lisp-callable graphics functions that map to renderer commands, managing rendering state.
3. **Renderer Wrapper**: Normalizes the renderer's API for Lisp integration, handling command execution and output.

The system avoids generating SVG XML, leveraging the renderer's direct image output. Lisp instructions (e.g., `(draw-line canvas x1 y1 x2 y2 stroke width)`) mirror SVG concepts, providing an intuitive abstraction for vector graphics.

3 Components

3.1 Lisp Interpreter Core

3.1.1 Purpose

The core handles parsing, evaluation, and environment management for Lisp code, serving as the runtime for graphics instructions.

3.1.2 Components

- **Tokenizer** (`tokenizer.py`): Converts Lisp code to tokens.
- **Parser** (`parser.py`): Builds abstract syntax trees (ASTs) from tokens.
- **Evaluator** (`evaluator.py`): Evaluates ASTs, supporting special forms and function calls.
- **Environment** (`environment.py`): Manages variable bindings and scopes.
- **Types** (`types.py`): Defines Lisp data types (e.g., Symbol, Pair, Number).
- **Main Interpreter** (`lisp.py`): Coordinates parsing and evaluation.
- **REPL** (`repl/console.py`): Provides an interactive interface for testing.

3.1.3 Role

The core remains unchanged, evaluating Lisp graphics commands and invoking functions defined in the graphics interface. It passes arguments to the renderer via the graphics module, unaware of rendering details.

3.2 Graphics Interface (`stdlib/graphics.py`)

3.2.1 Purpose

Exposes SVG-like vector graphics operations as Lisp-callable functions, mapping to renderer commands and managing rendering state.

3.2.2 Responsibilities

- Define functions (e.g., `make-canvas`, `draw-line`, `save-image`) as `NativeProcedure` instances.
- Manage graphics state (documents, canvases, shapes) to track rendering contexts.
- Validate arguments and handle errors using the interpreter's `LispError` hierarchy.
- Delegate commands to the renderer wrapper.

3.2.3 State Management

Maintains a `graphics_state` dictionary:

- **Documents**: `{document_id: {title, canvas_ids}}` groups canvases.
- **Canvases**: `{canvas_id: {document_id, renderer, width, height, viewBox, shapes}}` holds renderer instances and attributes.
- **Shapes**: `{shape_id: {canvas_id, type, attributes}}` tracks drawn elements (e.g., line, rectangle).

3.2.4 Key Functions

- `(make-document title)`: Creates a document for grouping canvases.
- `(make-canvas document-id width height)`: Initializes a canvas with a renderer instance.
- `(draw-line canvas-id x1 y1 x2 y2 stroke width)`: Draws a line.
- `(draw-rectangle canvas-id x y width height stroke fill width)`: Draws a rectangle.
- `(draw-circle canvas-id cx cy r stroke fill width)`: Draws a circle.
- `(clear-canvas canvas-id)`: Clears the canvas.
- `(save-image document-id filename)`: Saves the rendered image.
- `(display-image document-id)`: Displays the image.

3.3 Renderer Wrapper (`stdlib/svg_renderer.py`)

3.3.1 Purpose

Normalizes the SVG renderer's API for Lisp integration, encapsulating rendering logic and output handling.

3.3.2 Responsibilities

- Initialize renderer instances with canvas dimensions.
- Map Lisp commands to renderer methods (e.g., `draw-line` to `add_line`).
- Handle renderer-specific errors, translating them to interpreter-compatible exceptions.
- Provide output methods for saving images or displaying content.

3.3.3 Interface

- **Initialization**: Create a renderer instance (e.g., `RendererWrapper(width, height)`).
- **Drawing**: Methods like `add_line`, `add_rectangle`, `add_circle`.
- **Clearing**: `clear()` to reset the canvas.
- **Output**: `save(filename, format)` for image output; `display()` for viewing.

3.4 Standard Library Integration (`stdlib/__init__.py`)

3.4.1 Purpose

Loads the graphics module into the interpreter's global environment, making graphics functions available to Lisp code.

3.4.2 Responsibilities

- Include `graphics.py` and `svg_renderer.py` in the standard library.
- Call `graphics.load(environment)` to register functions.

4 Data Flow

1. **Lisp Execution**: The interpreter parses and evaluates Lisp code (e.g., `(draw-line canvas 10 10 100 100 "blue" 2)`), resolving functions to `NativeProcedure` instances in `graphics.py`.
2. **Graphics Processing**: `graphics.py` validates arguments, updates `graphics_state`, and calls the renderer wrapper's corresponding method (e.g., `add_line`).
3. **Rendering**: The wrapper invokes the renderer's method, updating its internal state or buffer.
4. **Output**: Commands like `(save-image)` or `(display-image)` trigger the renderer to produce an image file or display the content.

5 Architectural Principles

- **Separation of Concerns**: The interpreter handles Lisp execution, `graphics.py` defines the Lisp interface, and `svg_renderer.py` encapsulates renderer logic.
- **Modularity**: The graphics module is isolated, requiring no core changes. The wrapper allows renderer swaps.
- **Extensibility**: New commands or features (e.g., shape modification) can be added without affecting the interpreter.
- **Abstraction**: Lisp instructions use SVG-like abstractions, hiding renderer details.

6 Error Handling

- **Interpreter Errors**: Syntax or runtime errors are handled by `errors.py`.
- **Graphics Errors**: Invalid inputs (e.g., canvas IDs) raise `LispError` in `graphics.py`.
- **Renderer Errors**: The wrapper translates renderer exceptions to `LispError` or `ValueError`.

7 Output and Display

- **Image Output**: The renderer produces images via `save-image`, using formats like PNG.
- **Display**: `display-image` uses the renderer's display method or opens a temporary image in a system viewer.
- **No SVG Generation**: The renderer directly produces images, bypassing SVG XML.

8 Example Workflow

Lisp code:

```
(define doc (make-document ”Drawing”))
(define canvas (make-canvas doc 400 300))
(draw-line canvas 10 10 100 100 ”blue” 2)
(draw-rectangle canvas 50 50 100 80 ”black” ”red” 1)
(draw-circle canvas 200 150 50 ”green” ”yellow” 2)
(save-image doc ”output.png”)
(display-image doc)
```

- The interpreter evaluates commands, calling `graphics.py` functions.
- `graphics.py` updates state and delegates to the renderer wrapper.
- The wrapper invokes renderer methods, building the image.
- The renderer saves or displays the final image.

9 Future Enhancements

- **Shape Manipulation**: Add functions to modify or delete shapes using `shape_id`.
- **Transformations**: Support SVG-style transformations (e.g., rotate, scale).
- **Interactive Graphics**: Integrate event handling for user input, if supported by the renderer.
- **Batch Rendering**: Buffer commands for optimized rendering.

10 Conclusion

This architecture provides a modular, extensible integration of a Lisp interpreter and SVG 1.1 renderer, aligning Lisp's functional model with the renderer's vector graphics capabilities. By isolating the renderer in a wrapper and exposing SVG-like abstractions in Lisp, the system ensures portability, maintainability, and flexibility for direct image rendering.