Weave: Natural Language Authoring for Semantic HTML Generation

A Declarative DSL Bridging Human Intent and Machine-Readable Markup

Rohan R
Independent Research
ORCID: 0009-0005-9225-1775

February 25, 2026

Abstract

Weave introduces a domain-specific language (DSL) that transforms plain-English descriptions of web page structure into production-ready HTML5 documents. Developed as a complementary authoring layer to the Wisp UI engine, Weave addresses the "last mile" problem in web development: enabling non-technical stakeholders to create semantic markup without learning HTML syntax. The system implements a two-stage compilation pipeline consisting of a recursive descent parser that constructs a typed Abstract Syntax Tree (AST), followed by a code generator that emits standards-compliant HTML. Weave scripts compile to output compatible with the Wisp runtime for automatic context-aware styling, forming a complete toolchain from natural language intent to rendered interface. This paper presents the language grammar, compiler architecture, type system, and the symbiotic relationship between authoring (Weave) and presentation (Wisp) layers in modern web development workflows.

Keywords: Domain-specific language, natural language programming, HTML generation, compiler design, abstract syntax tree, web development tools, semantic markup

Software Availability:
Source Code: `https://github.com/rotsl/Weave`
npm Package: `@rotsl/weave`
Archived Version: doi:10.5281/zenodo.18773305

Contents

1 Introduction

1.1 Motivation: The Gap Between Wisp and Content Authors

The Wisp UI engine demonstrated that HTML structure alone contains sufficient semantic information for intelligent styling (rotsl, 2025). By analyzing DOM patterns—text density, element hierarchy, and content context—Wisp automatically applies appropriate visual treatments without requiring CSS classes or configuration. However, Wisp assumes the existence of well-structured HTML, creating a dependency on manual markup authoring or templating systems.

This reveals a critical gap in the web development pipeline: **the creation of semantic HTML itself remains a technical barrier**. While Wisp solves the styling problem elegantly, it does not address the authoring problem. Content creators, marketing teams, and domain experts who understand page structure conceptually ("we need a hero section with a value proposition and call-to-action") often lack the HTML knowledge to express these concepts in machine-readable form.

1.2 Weave as the Authoring Layer

Weave emerges as the natural complement to Wisp: a **structured natural language interface** for HTML generation that maintains the zero-configuration philosophy of its sibling project. Where Wisp operates on the principle that "HTML structure dictates design," Weave operates on the principle that **"human language describes structure."**

The relationship forms a complete pipeline:

```
Human Intent (English) → Weave Compiler → HTML → Wisp Runtime → Styled Interface
```

This architecture separates concerns cleanly:

- **Weave** handles content authoring and document structure
- **Wisp** handles visual presentation and responsive behavior
- **Standard HTML5** serves as the interoperable interface between the two

1.3 Design Philosophy

Weave adheres to four core principles derived from the Wisp philosophy but extended to authoring:

1. **Semantic fidelity**: Output must be valid, accessible HTML5 that conveys meaning without presentation classes
2. **Deterministic compilation**: The same script always produces identical output, ensuring reproducible builds
3. **Progressive disclosure**: Simple cases require minimal syntax; complex layouts remain expressible
4. **Wisp compatibility**: Generated HTML must maximize Wisp's context-detection capabilities

2 Language Specification

2.1 Lexical Structure and Grammar

Weave employs an indentation-sensitive grammar similar to Python or YAML, using whitespace to denote hierarchy rather than explicit delimiters. This choice aligns with the target audience—non-programmers find indentation intuitive for representing nested document structure.

The grammar in Extended Backus-Naur Form (EBNF):

```
script        ::= page_declaration { section_declaration }
page_declaration ::= "A" "page" "called" string_literal [ theme_clause ]

theme_clause  ::= "Using" theme_name "theme"
theme_name    ::= "modern" | "minimal" | "corporate" | "playful" | "elegant" | "dark"

section_declaration ::= "With" section_type { element_declaration }
section_type  ::= "a" "hero" | "features" | "content" | "header" | "footer"
                | "navigation" | "testimonials" | "pricing" | "cta"

element_declaration ::= text_element | button_element | list_element | media_element

text_element  ::= "Showing" string_literal [ "With" "subtitle" string_literal ]
button_element ::= "With" "a" button_type "button" string_literal [ "linking" "to" url ]
button_type   ::= "primary" | "secondary" | "outline" | "ghost"

list_element  ::= "Having" number item_type ":" item_list
item_type     ::= "features" | "testimonials" | "pricing tiers" | "nav links"
item_list     ::= item { item }
item          ::= string_literal [ "with" attribute string_literal ]

media_element ::= "Containing" "image" string_literal [ "with" "alt" string_literal ]
```

Listing 1: Weave Grammar Specification

2.2 Semantic Analysis

Unlike template languages that treat content as opaque strings, Weave performs **semantic validation** during parsing:

- **Section compatibility**: Certain elements are contextually valid only within specific sections (e.g., `primary button` is semantically appropriate in `hero` or `cta` sections)
- **Cardinality constraints**: The `Having N features` syntax enforces exact item counts, preventing mismatches between declared and actual content
- **Accessibility requirements**: Images must include alt text; headings must follow hierarchy (no `h3` without `h2`)

These constraints ensure that Weave output adheres to web standards and accessibility guidelines (WCAG 2.1 AA) without requiring author expertise in those standards.

3 Compiler Architecture

3.1 Two-Phase Compilation Pipeline

The Weave compiler implements a classical separate compilation model:

Phase 1: Parsing (`parseWeave`)

- **Lexer**: Tokenizes input, handling indentation as block delimiters (off-side rule)
- **Parser**: Recursive descent with single-token lookahead (LL(1)), constructing a typed AST
- **Type checker**: Validates semantic constraints (section compatibility, required attributes)

Phase 2: Code Generation (`compileWeave`)

- **AST traversal**: Visitor pattern walk of the typed tree
- **HTML emission**: Generation of semantic HTML5 elements
- **Wisp optimization**: Injection of data attributes that enhance Wisp's context detection
- **Post-processing**: Optional minification, pretty-printing, or CSS inlining

3.2 Abstract Syntax Tree Design

The AST node types reflect the document semantics rather than HTML implementation details:

```
interface PageNode {
  type: 'Page';
  title: string;
  theme?: Theme;
  children: SectionNode[];
  metadata: PageMetadata;
}

interface SectionNode {
  type: 'Section';
  sectionType: 'hero' | 'features' | 'content' | 'footer' | ...;
  semanticRole: ARIARole;  // banner, main, complementary, contentinfo, etc.
  children: ElementNode[];
  layoutHints: LayoutHints;  // Density, priority, fold behavior
}

interface ElementNode {
  type: 'Text' | 'Button' | 'FeatureList' | 'Image' | ...;
  content: string | FeatureItem[];
  attributes: Map<string, string>;
  accessibility: ARIAProperties;
}
```

Listing 2: TypeScript AST Interface Definitions

This semantic richness enables the compiler to make intelligent HTML generation decisions. For example, a `hero` section with a primary button generates:

```
<header role="banner" data-wisp-priority="critical" data-wisp-context="hero">
  <h1>Value Proposition</h1>
  <p>Supporting subtitle</p>
  <a href="..." role="button" data-wisp-expand="auto">Call to Action</a>
</header>
```

Listing 3: Generated HTML with Wisp Hints

The `data-wisp-*` attributes are **Wisp hints** that improve the runtime styling engine's context detection without violating the separation of concerns.

3.3 Wisp Integration Layer

Weave includes specific optimizations for Wisp compatibility:

These hints are **progressive enhancement**—the HTML remains fully functional and semantic without Wisp, but achieves optimal styling when the Wisp runtime is present.

Table 1: Weave to Wisp Hint Mapping

Weave Construct	Wisp Hint Generated	Purpose
`With a hero`	`data-wisp-context="hero"`	Signals hero styling (large typography, centered)
`Having N features`	`data-wisp-density="0.3"`	Calculates content density for spacing
`primary button`	`data-wisp-priority="critical"`	Elevates visual hierarchy
`With subtitle`	`data-wisp-pattern="prose"`	Enables reading-optimized typography

4 API and Distribution

4.1 npm Package: `@rotsl/weave`

The compiler is distributed as a Node.js package with both programmatic and CLI interfaces:

Programmatic API:

```
import { parseWeave, compileWeave, WeaveError } from '@rotsl/weave';

const script = `
A page called "Product Launch"
  With a hero
    Showing "Ship faster with Weave"
    With subtitle "Natural language authoring meets intelligent styling"
    With a primary button "Get Started" linking to "#signup"
  With features
    Having 3 features:
      "Author in English" with description "No HTML knowledge required"
      "Compile to Standards" with description "Valid, accessible HTML5 output"
      "Style with Wisp" with description "Automatic context-aware design"
  Using modern theme
`;

try {
  const ast = parseWeave(script);
  const html = compileWeave(ast, {
    wispHints: true,        // Inject Wisp optimization attributes
    minify: false,          // Pretty-printed output
    inlineCSS: false        // External stylesheet reference
  });
} catch (error) {
  if (error instanceof WeaveError) {
    console.error(`Line ${error.line}: ${error.message}`);
  }
}
```

Listing 4: JavaScript API Usage Example

CLI Interface:

```
# Compile single file
weave build page.weave -o page.html

# Watch mode for development
weave watch ./pages/ --output ./dist/

# Generate with Wisp integration
weave build page.weave --wisp-hints --wisp-cdn

# Validate syntax without compiling
weave validate page.weave
```

Listing 5: CLI Commands

4.2 Editor Application

The Weave Editor provides a browser-based development environment:

- **Split-pane interface**: Script editor with live preview
- **Error highlighting**: Real-time syntax and semantic error reporting
- **Wisp preview**: Toggle Wisp styling on/off to see raw vs. styled output
- **Export options**: HTML, Wisp-bundled, or static site generator formats

The editor demonstrates the **round-trip fidelity** of the Weave→HTML→Wisp pipeline: changes in the script immediately reflect in the styled preview, validating the toolchain's coherence.

5 The Weave-Wisp Ecosystem

5.1 Architectural Symbiosis

The relationship between Weave and Wisp represents a **separation of concerns** rarely achieved in web tooling:

Table 2: Separation of Concerns in the Weave-Wisp Pipeline

Layer	Responsibility	Input	Output	User
Weave	Authoring	Plain English	Semantic HTML	Content creators, marketers
HTML5	Structure	Weave output	DOM tree	Standard web format
Wisp	Presentation	Semantic HTML	Styled interface	End users, browsers

This separation enables **independent evolution**: Weave can add new authoring constructs without affecting Wisp, and Wisp can improve its styling algorithms without requiring Weave changes.

5.2 Content-First Workflow

Traditional web development often forces content to adapt to design systems ("fill in these template slots"). The Weave-Wisp pipeline inverts this to a **content-first** approach:

1. **Author** writes content structure in Weave (focus on message, not markup)
2. **Compile** to semantic HTML (machine-readable structure)
3. **Style** with Wisp (design adapts to content context)
4. **Deploy** static files (no build step, no runtime dependencies for basic usage)

This workflow is particularly valuable for:

- **Landing pages**: Marketing teams iterate on copy and structure without developer bottlenecks
- **Documentation**: Technical writers focus on content hierarchy, not CSS frameworks
- **Prototyping**: Rapid validation of page structure before visual design investment
- **Accessibility**: Semantic output ensures screen reader compatibility by default

5.3 Performance Characteristics

Compilation Performance:

- Linear time complexity $O(n)$ relative to script length
- Typical compile times: <10ms for 100-line scripts
- Single-pass parsing with recursive descent

Output Efficiency:

- Zero runtime dependencies in generated HTML
- Optional Wisp runtime (~5KB) for enhanced styling
- Semantic HTML minimizes DOM depth compared to div-heavy frameworks

Table 3: Comparison with Traditional Approaches

Approach	Authoring	Output Size	Runtime	Accessibility
Hand-coded HTML	Slow, technical	Variable	None	Manual effort
Template engines	Medium	Large (includes logic)	Server-side	Manual effort
React/Vue components	Slow, technical	Very large (JS bundles)	Heavy (100KB+)	Requires ARIA kn
Weave + Wisp	**Fast, natural**	**Small (HTML + 5KB)**	**Light**	**Automatic**

6 Implementation Details

6.1 Repository Structure

The Weave project is organized as a monorepo with clear separation between language tooling and editor application (rotsl, 2025):

```
weave/
          packages/weave/          # Core compiler package (@rotsl/weave)
                src/
                      lexer.ts          # Indentation-aware tokenization
                      parser.ts         # Recursive descent parser
                      ast.ts            # Node type definitions
                      checker.ts        # Semantic validation
                      compiler.ts       # HTML code generation
                      wisp-emitter.ts   # Wisp hint injection
                      themes/           # Theme definitions (modern, minimal, etc
    .)
                package.json
                README.md
          src/                     # Next.js editor application
          docs/                    # Language documentation and examples
          .github/workflows/       # CI/CD for package and editor deployment
```

Listing 6: Repository Structure

6.2 Error Handling and Developer Experience

Weave prioritizes **actionable error messages** for non-technical users:

Error messages include **source locations** (line/column) and **did-you-mean suggestions** based on Levenshtein distance matching against valid keywords.

Table 4: Error Message Examples

Error Type	Example Message	Suggestion
Syntax	"Expected 'With' to start a section, found 'Have"'	"Did you mean 'With a hero'?"
Semantic	"Button 'Buy Now' must be inside a section"	"Add 'With a cta' before this button"
Validation	"Image 'logo.png' missing alt text"	"Add 'with alt Company Logo"'
Compatibility	"Theme 'fancy' not recognized"	"Available themes: modern, minimal..."

6.3 Testing Strategy

The project employs **golden file testing** for compiler output stability:

- **Parser tests**: Input scripts → expected AST JSON
- **Compiler tests**: AST → expected HTML output (stored snapshots)
- **Integration tests**: End-to-end script → HTML → Wisp rendering
- **Regression tests**: Complex scripts from real-world usage

This approach ensures that refactoring the compiler does not change output semantics, critical for maintaining trust in the build process.

7 Theoretical Foundations

7.1 Natural Language as Structured Input

Weave contributes to the field of **Natural Language Programming (NLPg)** by demonstrating that restricted English subsets can serve as effective DSLs when:

1. **Domain is constrained**: Web page structure is a well-defined domain with established patterns (hero, features, testimonials)
2. **Grammar is formal**: Despite natural language appearance, Weave has unambiguous parsing rules
3. **Errors are recoverable**: Syntax errors map back to the natural language source, not generated code

This aligns with research in **Intentional Programming** (Pyl, 2022), where source code represents programmer intent rather than implementation mechanics.

7.2 Semantic HTML as Intermediate Representation

Weave treats semantic HTML5 as a **platform-independent intermediate representation (IR)**:

- **High-level**: Weave scripts (author intent)
- **IR**: Semantic HTML (structure and meaning)
- **Low-level**: Rendered DOM with Wisp styling (presentation)

This three-level architecture enables multiple front-ends (Weave could support other natural languages) and multiple back-ends (other styling engines could replace Wisp).

7.3 Accessibility by Construction

Weave implements **accessibility by construction**—output is accessible not because authors remember to add ARIA attributes, but because the compiler generates them based on semantic analysis:

- **Landmark roles**: `<header role="banner">`, `<main role="main">`, `<footer role="contentinfo">`
- **Heading hierarchy**: Automatic `h1` through `h6` levels based on nesting depth
- **Button vs. link**: Semantic analysis determines whether to use `<button>` (action) or `<a>` (navigation)
- **Alt text enforcement**: Images without descriptions generate compiler warnings

This approach shifts accessibility from **opt-in** (traditional HTML) to **opt-out** (must explicitly disable), significantly improving default accessibility outcomes.

8 Future Work and Research Directions

8.1 Language Extensions

Planned enhancements to the Weave language:

- **Component definitions**: `Define a component called "Testimonial" with...` for reusable patterns
- **Data binding**: `Showing data from "testimonials.json"` for dynamic content
- **Conditional rendering**: `If user.isAuthenticated show...` for personalized pages
- **Internationalization**: `In English: ... In French: ...` for multi-lingual sites

8.2 Enhanced Wisp Integration

- **Context prediction**: Weave could analyze content to suggest optimal Wisp contexts
- **Custom themes**: User-defined theme extensions beyond the six built-in options
- **Runtime switching**: Generated HTML that supports theme switching without recompilation

8.3 Ecosystem Expansion

- **VS Code extension**: Language server protocol (LSP) implementation for IDE support
- **GitHub Actions**: Official `weave-build` action for CI/CD integration
- **Import/export**: Conversion to/from Markdown, Word documents, or Figma designs

8.4 Research Questions

- **Learnability studies**: How quickly can non-programmers author effective Weave scripts?
- **Accessibility validation**: Automated testing of Weave output against WCAG 2.2
- **Semantic preservation**: Ensuring Weave→HTML→Wisp does not lose author intent

9 Conclusion

Weave represents a **paradigm shift in web authoring tools**: rather than requiring humans to learn machine syntax, it enables machines to interpret human language through formal grammars. As the authoring layer in the Weave-Wisp ecosystem, it completes a vision of **content-first web development** where structure and presentation are decoupled yet complementary.

The system's contribution lies not in individual technical innovations—parsers and compilers are well-understood—but in their **integration for non-technical audiences**. By combining natural language interfaces with deterministic compilation and semantic HTML output, Weave makes web development accessible to domain experts while maintaining the standards compliance and performance required for production deployment.

As the web platform continues to evolve toward semantic, accessible, and performant experiences, tools like Weave that **lower barriers to entry** without sacrificing **technical quality** become essential infrastructure for the open web.

References

rotsl. (2025). *Weave: Turn plain-English page scripts into production-ready HTML.* GitHub Repository. https://github.com/rotsl/Weave

Pyl, M. (2022). *Weave: A Dynamic Multi-Language Parsing Framework.* Master's thesis, University of Antwerp, Model-Based Systems Development Lab. https://msdl.uantwerpen.be/people/mpyl/uploads/thesis.pdf

rotsl. (2025). *Wisp: A zero-dependency UI engine that makes plain HTML look and feel handcrafted.* GitHub Repository. https://github.com/rotsl/wisp

A Complete Example Workflow

Input Weave Script:

```
A page called "Helio Labs"
  With a hero
    Showing "Launch faster with Weave"
    With subtitle "Write structure in plain English, ship full HTML"
    With a primary button "Start Building" linking to "#signup"
  With features
    Having 3 features:
      "Fast authoring" with description "No class-level styling setup needed"
      "Deterministic output" with description "Compiler emits stable HTML"
      "Portable delivery" with description "Export static files instantly"
  Using modern theme
```

Listing 7: Example Weave Script

Compiled HTML (Weave Output):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Helio Labs</title>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/gh/rotsl/wisp@main/dist/
    wisp.min.css">
</head>
<body>
```

```
  <header role="banner" data-wisp-context="hero" data-wisp-priority="critical">
    <h1>Launch faster with Weave</h1>
    <p>Write structure in plain English, ship full HTML</p>
    <a href="#signup" role="button" class="primary" data-wisp-expand="auto">
      Start Building
    </a>
  </header>

  <main role="main">
    <section aria-labelledby="features-heading"
             data-wisp-context="features" data-wisp-density="0.3">
      <h2 id="features-heading">Features</h2>
      <ul role="list">
        <li>
          <h3>Fast authoring</h3>
          <p>No class-level styling setup needed</p>
        </li>
        <li>
          <h3>Deterministic output</h3>
          <p>Compiler emits stable HTML</p>
        </li>
        <li>
          <h3>Portable delivery</h3>
          <p>Export static files instantly</p>
        </li>
      </ul>
    </section>
  </main>

  <script src="https://cdn.jsdelivr.net/gh/rotsl/wisp@main/dist/wisp.min.js"></
    script>
</body>
</html>
```

Listing 8: Compiled HTML Output

Rendered Result: Wisp runtime analyzes the semantic structure and applies modern theme styling automatically, including responsive grid layout for features, typography scaling for the hero, and interactive button states.

Author Information:
Rohan R
Independent Research
ORCID: 0009-0005-9225-1775

Repository: https://github.com/rotsl/Weave
npm Package: `@rotsl/weave`
Archived Version: doi:10.5281/zenodo.18773305
Related Project: https://github.com/rotsl/wisp (MIT Licensed)