1. Architectural Vision: Transformation and Generation in a Zero-Copy World

The Minotaur platform is founded on a high-performance, zero-copy architecture where the CognitiveGraph serves as an immutable, in-memory representation of source code, complete with its syntactic ambiguities and semantic richness. Project Golem extends this foundation by introducing capabilities for code correction and transformation. This requires two new, critical components:

1. **The CognitiveGraph Editor:** A component responsible for applying modifications to the graph. Given the immutable, zero-copy nature of the CognitiveGraph, this cannot be a direct, in-place mutation. Instead, the Editor orchestrates the creation of a *new* graph that incorporates the desired changes.
2. **The Unparser:** A component responsible for the reverse of parsing—generating human-readable source code from a CognitiveGraph. This is essential for materializing the results of any correction or translation.

This specification details the design of these two components, ensuring they integrate seamlessly with Minotaur's existing step-lexer, step-parser, and zero-copy principles.

---

2. The CognitiveGraph Editor: A Strategy for Zero-Copy Modification

Directly modifying the CognitiveGraph's memory buffer is not feasible. Any change in the length or structure of a code segment would invalidate all subsequent byte offsets in the buffer, corrupting the entire graph. The GraphEditor therefore implements a **"copy-on-write with incremental re-parse"** strategy, which is both safe and highly performant.

2.1. Core Component: The GraphEditor Class

The GraphEditor is the primary C# class for orchestrating all graph modifications. It leverages the existing Minotaur parser to ensure that all transformations result in a valid CognitiveGraph.

C#

```
/// <summary>
/// Orchestrates the modification of an immutable, zero-copy CognitiveGraph.
```

```
/// </summary>
public class GraphEditor
{
    private readonly MinotaurParser _parser;

    public GraphEditor(MinotaurParser parser)
    {
        _parser = parser;
    }

    /// <summary>
    /// Replaces a region of the graph corresponding to a node with new source text.
    /// </summary>
    /// <param name="originalGraph">The source CognitiveGraph.</param>
    /// <param name="nodeToReplace">The node identifying the region to be
replaced.</param>
    /// <param name="newSourceText">The new source code for the region.</param>
    /// <returns>A new, valid CognitiveGraph incorporating the change.</returns>
    public CognitiveGraph Replace(
        CognitiveGraph originalGraph,
        GraphNode nodeToReplace,
        string newSourceText)
    {
        // Implementation follows the three-phase process below.
    }
}
```

2.2. The Three-Phase Transformation Process

The Replace method executes a precise, three-phase workflow that respects the zero-copy architecture while efficiently applying changes.

Phase 1: Source Text Splicing

The transformation begins at the source of truth: the UTF-8 text buffer. A new source text buffer is created in memory that reflects the intended modification.

1. **Identify Edit Region:** The nodeToReplace accessor provides the SourceStart and SourceEnd byte offsets from the original graph's buffer.
2. **Construct New Source Buffer:** A new Memory<byte> buffer is assembled by combining three segments:

    - A slice of the original source buffer from the beginning up to SourceStart.
    - The UTF-8 byte representation of the newSourceText.
    - A slice of the original source buffer from SourceEnd to the end.

This step creates the new target source code in memory, ready for parsing.

Phase 2: Incremental Re-Parsing with Minotaur

This phase is the core performance optimization. Instead of re-parsing the entire new source buffer, the GraphEditor invokes Minotaur's parser in an **incremental mode**. This reuses the vast majority of the work already done in the originalGraph.

1. **Parser Invocation:** The Minotaur parser is called with both the newSourceBuffer and the originalGraph.
2. **Reuse of Unchanged Subgraphs:** The parser's engine compares the new buffer against the old one. It recognizes that the regions before and after the edit are identical. For these sections, it does not re-parse; instead, it directly reuses the corresponding subgraphs from the originalGraph. This is a fundamental capability of incremental parsing systems.[1]
3. **Delta Parsing:** The parser invalidates only the nodes within the edited region. It then invokes the step-lexer and step-parser exclusively on the new text segment. This generates a new, temporary in-memory Shared Packed Parse Forest (SPPF) for just the changed code, correctly capturing its syntax and any new ambiguities.
4. **Graph Stitching:** The newly generated SPPF subgraph is "stitched" together with the reused subgraphs from the original CognitiveGraph, forming a complete, temporary in-memory object graph (TempSymbolNode hierarchy) that represents the updated file.

This incremental approach ensures that the parsing time is proportional to the size of the edit, not the size of the file, making it suitable for real-time interactive scenarios.

Phase 3: Zero-Copy Serialization and Semantic Re-linking

The final phase converts the temporary object graph into a new, immutable CognitiveGraph buffer and ensures its semantic integrity.

1. **Serialization:** The GraphBuilder component performs its standard post-order traversal on the temporary graph, serializing it into a new, final Memory<byte> buffer. This process correctly calculates and writes all the new offsets required for the updated structure.
2. **Semantic Trigger Re-evaluation:** After the new graph buffer is created, a semantic linking pass is executed. This pass focuses on the "seams" between the reused subgraphs and the newly generated subgraph. It re-evaluates the semantic triggers for the new and adjacent nodes to create and update the CPG overlay (e.g., CONTROL_FLOW, DATA_FLOW, CALLS edges), ensuring the graph's semantic information is consistent across the entire project context.

The result of this three-phase process is a new, fully valid, and performance-optimized CognitiveGraph that accurately reflects the code modification.

---

3. The Unparser: Generating Code from the Cognitive Graph

Generating source code from a CognitiveGraph is the reverse of parsing and requires a dedicated **Unparser** component. It cannot reuse the step-parser or step-lexer, as their function is recognition, not generation. The Unparser's primary role is to traverse a graph and emit a well-formatted text file.

3.1. Prerequisite: Disambiguation

The CognitiveGraph can represent a forest of possible parse trees. To generate a single, valid source file, a specific parse tree must be chosen. This is a critical prerequisite for unparsing.

- **Responsibility:** The Golem engine, before calling the Unparser, must resolve all ambiguities in the CognitiveGraph. This can be done using various strategies (heuristics, rule-based logic, or selecting the highest-ranked GenAI solution).
- **Input to Unparser:** The Unparser does not operate on the raw CognitiveGraph. Instead, it operates on a **disambiguated traversal path**—a specific, acyclic route through the graph that represents a single, coherent Abstract Syntax Tree (AST).

3.2. The Unparsing Process

The Unparser uses a grammar-driven, recursive traversal to generate the source code.

1. **Grammar-Driven Templates:** The Unparser's logic is derived from the target language's .grammar file. For each rule in the grammar, there is a corresponding template for how to emit its syntax. For example, the grammar rule if_statement: 'if' '(' expression ')' statement; informs the Unparser to:
    - Emit the if keyword.
    - Emit a space.
    - Emit an opening parenthesis (.
    - Recursively call the Unparser on the expression child node.
    - Emit a closing parenthesis).
    - Recursively call the Unparser on the statement child node.
2. **Recursive Traversal:** The Unparser is implemented using a Visitor pattern, with a Visit method for each SymbolNode type. It performs a depth-first traversal of the disambiguated AST path, emitting tokens and recursively visiting child nodes according

to the grammar templates.

3. **Formatting and Pretty-Printing:** The CognitiveGraph is an *abstract* representation and does not store non-essential information like whitespace or indentation. The Unparser is responsible for re-introducing this formatting to produce human-readable code.
    - **Indentation:** The Unparser maintains the current indentation level, increasing it when entering a new block scope (e.g., a method body) and decreasing it upon exit.
    - **Whitespace and Newlines:** The grammar-driven templates specify where whitespace and newlines are conventional (e.g., after a semicolon, around operators).
    - **Comments:** The Unparser can also be designed to re-attach comments to their nearest nodes, preserving documentation. For C#, this can be achieved with high fidelity using Roslyn's formatting APIs like Formatter.Format or SyntaxNode.NormalizeWhitespace().[3] A similar, grammar-aware engine would be used for other languages within Minotaur.

3.3. Unifying Full and Partial Code Generation

The same Unparser component is used for both full-file transpilation and partial code correction.

- **For Code Translation:** The Golem engine first transforms the entire source CognitiveGraph into a new CognitiveGraph representing the target language. The Unparser is then invoked on the root node of this new graph to generate the complete target file.
- **For Code Correction:** The GraphEditor.Replace method produces a new, complete CognitiveGraph for the *same language*. The Unparser is invoked on the root of this modified graph to generate the full, corrected source file.

By separating the concerns of graph modification (GraphEditor) and text generation (Unparser), the system remains modular, robust, and aligned with the high-performance, zero-copy principles of the Minotaur platform.

Works cited

1. Tree-sitter: Introduction, accessed September 4, 2025, https://tree-sitter.github.io/
2. Python bindings to the Tree-sitter parsing library - GitHub, accessed September 4, 2025, https://github.com/tree-sitter/py-tree-sitter
3. Generating C# code with Roslyn APIs - Jeremy Davis, accessed September 4, 2025, https://blog.jermdavis.dev/posts/2024/csharp-code-with-roslyn
4. Generating well-formatted syntax with Roslyn - Stack Overflow, accessed September 4, 2025, https://stackoverflow.com/questions/33467295/generating-well-formatted-syntax

-with-roslyn